PROGRAMMES ET ALGORITHMES
Partie II

B. Landelle

Table des matières

I Recherche séquentielle

1 Recherche du minimum/maximum

• Recherche du maximum

La fonction `maxi(L)` d'argument `L` une liste non vide de nombres renvoie le maximum de cette liste.

```
def maxi(L):
    """maxi(L:list)->int or float
    Renvoie le maximum de L"""
    res=L[0]
    for x in L:
        if res<x:
            res=x
    return res
```

On initialise la variable `res` avec le premier élément de la liste `L`. On parcourt ensuite directement la liste `L` avec la variable locale `x` et si on rencontre un élément plus grand que celui stocké dans `res`, alors on garde la mémoire dans `res` de cet élément. En sortie de boucle, on a donc conservé le plus grand élément.

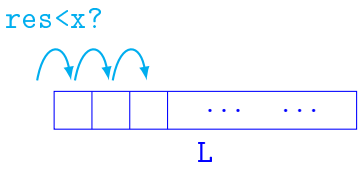

FIGURE 1 – Recherche de maximum

```
>>> a=[1,5,2,3]
>>> maxi(a)
5
```

Avec la convention $\max \varnothing = -\infty$, on peut proposer une implémentation cohérente même pour une liste vide :

```
def maxi(L):
    """Maximum de L"""
    res=float('-inf')
    for x in L:
        if res<x:
            res=x
    return res
```

• Recherche du maximum avec position dans la liste

La fonction `maxi_pos(L)` d'argument `L` une liste non vide de nombres renvoie le maximum de cette liste et l'indice d'une occurrence de ce maximum.

```
def maxi_pos(L):
    """maxi_pos(L:list)->(int or float, int)
    Renvoie le maximum de L et l'indice d'une occurrence de ce maximum"""
    n=len(L)
    res,ind=L[0],0
    for k in range(n):
        if res<L[k]:
            res,ind=L[k],k
    return res,ind
```

Pour garder la mémoire d'une position d'une occurrence du maximum dans une liste, on réalise un parcours de la liste par indice. On initialise deux variables : `res` avec le premier élément de la liste et `ind` l'indice de l'élément stocké dans `res`. On parcourt ensuite avec la variable locale `k` toutes les valeurs d'indice de 0 à $n-1$ et si on rencontre un élément `L[k]` plus grand que celui stocké dans `res`, alors on garde la mémoire dans `res` de cet élément ainsi que son indice dans `ind`. En sortie de boucle, on a le maximum et un indice où celui-ci est atteint.

```
>>> a=[1,5,2,3]
>>> maxi_pos(a)
(5, 1)
```

Avec la convention $\max \varnothing = -\infty$, on peut proposer l'implémentation cohérente pour une liste vide :

```
def maxi_pos(L):
    n=len(L)
    res,ind=float('-inf'),None
    for k in range(n):
        if res<L[k]:
            res,ind=L[k],k
    return res,ind
```

Exercice : Quelle occurrence du maximum la fonction `maxi_pos` renvoie-t-elle : la première, la dernière, aucune des deux ? Modifier, si besoin, la fonction pour que celle-ci renvoie la dernière occurrence du maximum.

Corrigé : Le test avec inégalité stricte fait qu'on ne réaffecte pas la variable `res` même si on rencontre une nouvelle occurrence du maximum. Par conséquent, la fonction `maxi_pos` renvoie la première occurrence du maximum. Pour obtenir la dernière occurrence, il suffit de changer l'inégalité stricte en large ce qui a pour effet de réaffecter la variable `res` à chaque occurrence du maximum. On saisit :

```
def maxi_pos(L):
    """maxi_pos(L:list)->(int or float, int)
    Renvoie le maximum de L et l'indice de sa dernière occurrence"""
    n=len(L)
    res,ind=L[0],0
    for k in range(n):
```

```
        if res<=L[k]:
            res,ind=L[k],k
    return res,ind
```

On observe :

```
>>> a=[1,5,2,3,5,1]
>>> maxi_pos(a)
(5, 4)
```

• Recherche du minimum

Pour le minimum, il suffit d'adapter le programme de recherche du maximum. Par exemple :

```
def mini(L):
    """mini(L:list)->int or float
    Renvoie le minimum de L"""
    res=L[0]
    for x in L:
        if res>x:
            res=x
    return res
```

Ici encore, avec la convention $\min \varnothing = +\infty$, on peut proposer une implémentation cohérente même pour une liste vide :

```
def mini(L):
    """Minimum de L"""
    res=float('inf')
    for x in L:
        if res>x:
            res=x
    return res
```

• Recherche simultanée du maximum et du minimum d'une liste de nombres

La fonction `maxi_mini(L)` d'argument `L` une liste non vide de nombres renvoie le minimum et le maximum de cette liste.

```
def maxi_mini(L):
    """maxi_mini(L:list)->(int or float,int or float)
    Renvoie le minimum et le maximum de L"""
    m,M=L[0],L[0]
    for x in L:
        if M<x:
            M=x
        if m>x:
            m=x
    return m,M
```

On initialise les variables `m` et `M` avec le premier élément de la liste `L`. On parcourt ensuite directement la liste `L` avec la variable locale `x`. Si on rencontre un élément plus grand que celui stocké dans `M`, alors on garde la mémoire dans `M` de cet élément. Si on rencontre un élément plus petit que celui stocké dans `m`, alors on garde la mémoire dans `m` de cet élément. En sortie de boucle, on a donc conservé le plus grand et le plus petit élément.

```
>>> a=[1,5,2,3]
>>> maxi_mini(a)
(1, 5)
```

Exercice : Écrire une fonction `maxi_mini_pos(L)` d'argument `L` une liste non vide de nombres et qui renvoie le maximum et le minimum de la liste ainsi que les indices des premières occurrences de ce maximum et minimum.

Corrigé : On saisit :

```
def maxi_mini_pos(L):
    """maxi_mini(L:list)->[[int or float,int],[int or float,int]]
    Renvoie le minimum et le maximum de L
    avec indice de leur première occurrence"""
    m,ind_m=L[0],0
    M,ind_M=L[0],0
    n=len(L)
    for k in range(n):
        if M<L[k]:
            M,ind_M=L[k],k
        if m>L[k]:
            m,ind_m=L[k],k
    return [[m,ind_m],[M,ind_M]]
```

On généralise la démarche vue pour le maximum avec conservation de l'indice où celui-ci est atteint.

```
>>> a=[1,5,2,3]
>>> maxi_mini_pos(a)
[[1, 0], [5, 1]]
```

2 Recherche d'un élément dans une liste

• Présence d'un élément dans une liste

La fonction `detect(elt,L)` d'argument `elt` un objet et `L` une liste renvoie `True` si `elt` est présent dans `L` et `False` sinon.

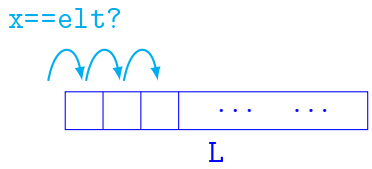

FIGURE 2 – Recherche d'un élément

Dans la version qui suit avec une boucle `for`, on remarque que l'on peut casser une boucle `for` avec un `return`. Ainsi, on sort de la boucle dès la première occurrence trouvée de `elt` tandis qu'on effectue toute la boucle si `elt` est absent de `L`.

```
def detect(elt,L):
    """detect(elt:any,L:list)->bool
    Renvoie test de présence de elt dans L"""
    for x in L:
        if x==elt:
            return True
    return False
```

On peut coder la même fonction avec une boucle conditionnelle.

```
def detect(elt,L):
    """detect(elt:any,L:list)->bool
    Renvoie test de présence de elt dans L"""
    k,n=0,len(L)
    while k<n and L[k]!=elt:
        k+=1
    return k<n
```

La variable `k` initialisée à 0 parcourt les indices des éléments de `L` jusqu'à rencontrer une occurrence de `elt` ou déborder de la plage $[\![0\,;n-1]\!]$ avec n la taille de `L`. Si on sort de la boucle lorsque `k==n`, c'est qu'on a parcouru toute la liste sans trouver l'élément ce qui signifie qu'il est absent. Ainsi, le test `k<n` en sortie fournit bien `True` si `elt` est dans `L` et `False` sinon.

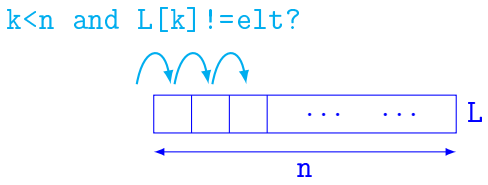

FIGURE 3 – Recherche d'un élément

Exercice : Peut-on coder la recherche d'un élément dans une liste de la manière suivante :

```
def detect(elt,L):
    k,n=0,len(L)
    while L[k]!=elt and k<n:
        k+=1
    return k<n
```

Justifier votre réponse.

Corrigé : Si `elt` est présent dans `L`, le code fonctionne de manière identique à la version précédente. En revanche, si `elt` est absent de `L`, on parcourt toute la liste et la variable `k` est donc incrémentée jusqu'à atteindre la valeur n, la taille de la liste. Or, l'accès `L[n]` provoque une erreur puisque les éléments de `L` sont indexés de 0 à $n-1$. La version précédente fonctionne

car elle utilise *l'évaluation paresseuse*. Si $k = n$, lors du test `k<n and L[k]!=elt`, le test `k<n` est `False` et python ne prend pas la peine d'évaluer la suite de l'expression après le `and` : il renvoie `False` d'où une sortie de la boucle sans message d'erreur.

• Première ou dernière occurrence d'un élément dans une liste

La fonction `occ(elt,L)` d'argument `elt` un objet et `L` une liste renvoie l'indice de la première occurrence de `elt` dans `L` si celui-ci est présent dans `L` et `len(L)` sinon. On parcourt la liste des indices avec un `range(n)` où n désigne la taille de la liste.

```
def occ(elt,L):
    """occ(elt:any,L:list)->int
    Renvoie indice première occurrence de elt dans L"""
    n=len(L)
    for k in range(n):
        if L[k]==elt:
            return k
    return n
```

La fonction renvoie la première occurrence de `elt` si celui-ci est présent dans `L` puisque dès qu'il est trouvé, on effectue un `return`. Pour avoir la dernière occurrence de `elt` dans `L`, on peut saisir :

```
def last_occ(elt,L):
    """last_occ(elt:any,L:list)->int
    Renvoie indice dernière occurrence de elt dans L"""
    n=len(L)
    for k in range(n-1,-1,-1):
        if L[k]==elt:
            return k
    return n
```

L'instruction `range(n-1,-1,-1)` génère la séquence $n-1$, $n-2$, ... 0 ce qui permet le parcours de `L` du dernier au premier indice. Le test provoquera donc la sortie pour la première occurrence de `elt` en partant de la fin, autrement dit pour la dernière occurrence de `elt` dans `L`.

Exercice : Écrire une version de `occ` et `last_occ` avec une boucle `while`.

Corrigé : On saisit :

```
def occ(elt,L):
    """occ(elt:any,L:list)->int
    Renvoie indice première occurrence de elt dans L"""
    k,n=0,len(L)
    while k<n and L[k]!=elt:
        k+=1
    return k

def last_occ(elt,L):
```

```
    """last_occ(elt:any,L:list)->int
    Renvoie indice dernière occurrence de elt dans L"""
    n=len(L)
    k=n-1
    while k>=0 and L[k]!=elt:
        k-=1
    if k<0:
        return n
    return k
```

• **Occurrences d'un élément dans une liste**

La fonction `pos(elt,L)` d'arguments `elt` un objet et `L` une liste renvoie une liste des indices de `elt` dans `L`. Si `elt` est absent de `L`, la fonction renvoie la liste vide.

```
def pos(elt,L):
    """pos(elt:any,L:list)->list
    Renvoie la liste des indices des occurrences de elt dans L"""
    res=[]
    n=len(L)
    for k in range(n):
        if L[k]==elt:
            res.append(k)
    return res
```

On parcourt la liste `L` par indice. À chaque fois qu'on rencontre `elt`, on garde son indice en mémoire en stockant celui-ci dans la variable `res`.

3 Recherche d'un mot dans un texte

On présente deux approches : l'une où l'on n'utilise que des opérations élémentaires, l'autre où l'on s'autorise à tester l'égalité sur des chaînes. Un test d'égalité sur des chaînes n'est pas une opération élémentaire : son temps de traitement est fonction de la taille des chaînes considérées.

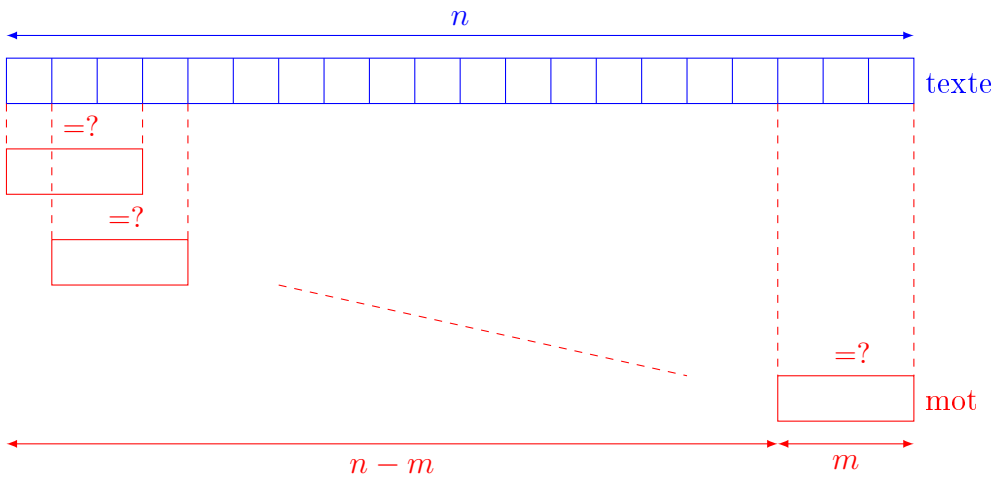

FIGURE 4 – Recherche d'un mot dans un texte

• **Recherche d'un mot avec opérations élémentaires**

On « externalise » la recherche d'un mot dans un texte à une position donnée avec la fonction `detect`.

```
def detect(elt,T,pos):
    """detect(elt:str,T:str,pos:int)->bool
    Renvoie le test elt==T[pos:pos+m] avec m=len(elt)"""
    for k in range(len(elt)):
        if elt[k]!=T[pos+k]:
            return False
    return True

def rech(mot,texte):
    """rech(mot:str,texte:str)->bool
    Renvoie le test de présence de mot dans texte"""
    m=len(mot)
    n=len(texte)
    for k in range(n-m+1):
        if detect(mot,texte,k):
            return True
    return False
```

La fonction `rech(mot,texte)` d'arguments `mot` et `texte` des chaînes renvoie `True` si `mot` est présent dans `texte` et `False` sinon. Dans la fonction `rech`, la variable `k` parcourt toutes les positions où peut démarrer le mot recherché.
On fournit à la fonction `detect` le mot recherché, le texte et l'indice où l'on cherche le mot. La fonction `detect(mot,T,pos)` parcourt le texte `T` sur la plage d'indices `pos` , ...,`pos+m-1` avec `m` la taille du mot et renvoie `True` si c'est le mot cherché, `False` sinon : dès qu'un caractère diffère, on sort et on renvoie `False`.

On peut coder cette recherche avec une boucle conditionnelle :

```
def rech(mot,texte):
    """rech(mot:str,texte:str)->bool
    Renvoie le test de présence de mot dans texte"""
    m=len(mot)
    n=len(texte)
    k=0
    while k<=n-m and not detect(mot,texte,k):
        k+=1
    return k<=n-m
```

Si le mot est présent dans le texte, le résultat renvoyé par `detect` sera vrai avant d'atteindre `k=n-m+1`. Si on atteint cette valeur, c'est qu'aucun résultat renvoyé par `detect` n'a permis la sortie de boucle. Par conséquent, le test `k<=n-m` renvoie bien un résultat d'appartenance de `mot` à `texte`.

Exercice : Peut-on coder la recherche d'un mot dans un texte de la manière suivante :

```
def rech(mot,texte):
    m=len(mot)
    n=len(texte)
    k=0
    while not detect(mot,texte,k) and k<=n-m:
        k+=1
    return k<=n-m
```

Justifier votre réponse.

Corrigé : Comme pour la détection d'un élément dans une liste, l'ordre pour ce test importe. Si le mot est absent du texte, la variable `k` peut atteindre la valeur `n-m+1` ce qui provoquera une erreur dans la fonction `detect`. Le test `k<=n-m and not detect(mot,texte,k)` fonctionne y compris pour `k` valant `n-m+1` grâce à l'évaluation paresseuse.

• Recherche d'un mot avec égalité de chaînes

En s'autorisant le test d'égalité entre chaînes, une version avec boucle inconditionnelle est donnée par :

```
def rech(mot,texte):
    """rech(mot:str,texte:str)->bool
    Renvoie le test de présence de mot dans texte"""
    m=len(mot)
    n=len(texte)
    for k in range(n-m+1):
        if mot==texte[k:k+m]:
            return True
    return False
```

et une version avec boucle conditionnelle :

```
def rech(mot,texte):
    """rech(mot:str,texte:str)->bool
    Renvoie le test de présence de mot dans texte"""
    m=len(mot)
    n=len(texte)
    k=0
    while k<=n-m and mot!=texte[k:k+m]:
        k+=1
    return k<=n-m
```

II Techniques importantes

1 Recherche dichotomique dans une liste triée

Quand on cherche un mot dans un dictionnaire, on l'ouvre *grosso modo* au milieu, on vérifie si on tombe sur le mot souhaité, sinon on choisit dans quelle moitié poursuivre la recherche et on

répète la démarche jusqu'au succès ou la certitude que le mot cherché n'y figure pas. Il s'agit d'un algorithme de *recherche dichotomique*.

Dans ce cours, on considère la situation de la recherche d'un entier dans une liste d'entiers triée mais ce cadre n'est pas exclusif : on peut appliquer ce qui suit pour toute liste d'éléments dans un ensemble muni d'une relation d'ordre total.

On cherche à savoir la position d'un objet `elt` dans une liste triée `L`. Le principe est le suivant :
- on considère l'élément au milieu de `L` ;
- si c'est `elt`, on s'arrête ;
- si `elt` est plut petit que l'élément du milieu, on se place sur la moitié de gauche, sinon on se place sur la moitié de droite ;
- on poursuit ce processus tant qu'on n'a pas rencontré `elt` et que la zone de recherche n'est pas vide.

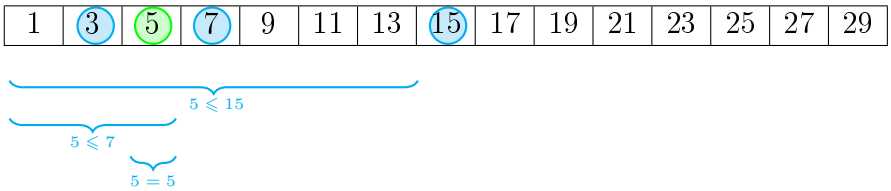

FIGURE 5 – Recherche dichotomique de 5 dans $[1, 3, \ldots, 29]$

```
def rech_dicho(elt,L):
    """rech_dicho(elt:int,L:list)->(bool,int)
    Renvoie le résultat de la recherche dichotomique
    de elt dans L liste triée :
    * si L[k]==elt       -> (True,k)
    * si elt absent de L -> (False,k)"""
    deb=0
    fin=len(L)-1
    trouve=False
    while not trouve and deb<=fin:
        milieu=(deb+fin)//2
        if L[milieu]==elt:
            trouve=True
        elif L[milieu]>elt:
            fin=milieu-1
        else:
            deb=milieu+1
    return trouve,milieu
```

Les variables `deb` et `fin` servent à délimiter la plage de recherche de `elt`. La variable booléenne `trouve` est l'indicateur de présence de `elt` dans `L`. La variable `milieu` reçoit la position milieu de la tranche de recherche. Tant que `trouve` est `False` et `deb<=fin`, c'est-à-dire `elt` non trouvé et plage de recherche non vide, on détermine le milieu de la zone de recherche, si `elt` est présent en `milieu`, on bascule la variable `trouve` à `True`, sinon, selon la position de `elt` par rapport à `milieu`, on modifie `deb` ou `fin` pour restreindre strictement la recherche à gauche ou à droite

du milieu.

On rend la fonction « bavarde » pour observer son fonctionnement :

```
...
    while not trouve and deb<=fin:
        milieu=(deb+fin)//2
        print(deb,fin,L[deb:fin+1])
...
```

On expérimente :

```
>>> L=list(range(1,30,2))
>>> L
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
>>> rech_dicho(7,L)
0 14 [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
0 6 [1, 3, 5, 7, 9, 11, 13]
(True, 3)
>>> rech_dicho(0,L)
0 14 [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
0 6 [1, 3, 5, 7, 9, 11, 13]
0 2 [1, 3, 5]
0 0 [1]
(False, 0)
>>> rech_dicho(29,L)
0 14 [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
8 14 [17, 19, 21, 23, 25, 27, 29]
12 14 [25, 27, 29]
14 14 [29]
(True, 14)
```

2 Coefficient binomial

Soit n entier et $k \in [\![0\,;\, n]\!]$. On a

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \begin{cases} \dfrac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1} & \text{si } k > 0 \\ 1 & \text{si } k = 0 \end{cases}$$

On en déduit l'implémentation suivante :

```
def binom(n,k):
    """binom(n:int,k:int)->int
    Renvoie le coefficient binomial k parmi n"""
    res=1
    for i in range(k):
        res=res*(n-i)//(i+1)
    return res
```

On initialise la variable `res` à 1. Lors du i-ème passage dans la boucle, la variable passe de $\binom{n}{i}$ à $\binom{n}{i+1}$ puisque

$$\binom{n}{i} \times \frac{n-i}{i+1} = \frac{n(n-1)\dots(n-i+1)}{i(i-1)\dots 1} \times \frac{n-i}{i+1} = \binom{n}{i+1}$$

Ainsi, après avoir effectué le produit `res*(n-i)`, la variable `res` contient

$$(n-i)\binom{n}{i} = (i+1)\binom{n}{i+1}$$

qui est divisible par $i+1$. On finalise le calcul en effectuant le quotient par $i+1$, ce quotient permettant d'avoir un résultat entier contrairement à une simple division.

3 Algorithme de Horner

Soit x un réel et $\mathrm{P} = \sum_{k=0}^{n-1} a_k \mathrm{X}^k \in \mathbb{R}[\mathrm{X}]$. Pour calculer efficacement $\mathrm{P}(x)$, on peut observer l'écriture suivante :

$$\mathrm{P}(x) = \sum_{k=0}^{n-1} a_k x^k = (\dots((a_{n-1}x + a_{n-2})\,x + a_{n-3})\,x + \dots)\,x + a_0$$

On en déduit l'implémentation de *l'algorithme de Horner* :

```
def poly(x,P):
    """poly(x:int or float,P:list)->int or float
    Calcul de P(x) suivant l'algorithme de Horner
    P : [a_0, ..., a_{n-1}] liste d'entiers ou flottants
    P code a_0+a_1*X+... a_{n-1}*X^(n-1)"""
    res=0
    n=len(P)
    for k in range(n-1,-1,-1):
        res=x*res+P[k]
    return res
```

Une application usuelle : Calcul d'un nombre à partir de son écriture binaire

Soit n entier non nul. Il existe un unique p entier non nul tel que

$$n = \sum_{k=0}^{p-1} d_k 2^k \quad \text{avec} \quad (d_0, \dots, d_{p-2}) \in \{0,1\}^{p-1} \quad \text{et} \quad d_{p-1} = 1$$

On note $$n = \langle d_{p-1}, d_{p-2}, \dots, d_0 \rangle$$

l'écriture binaire de n, à savoir son écriture en base deux. Par exemple, on a

$$2 = 1 \times 2^1 + 0 \times 2^0 = \langle 1, 0 \rangle \qquad 10 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = \langle 1, 0, 1, 0 \rangle$$

et $$123 = \langle 1, 1, 1, 1, 0, 1, 1 \rangle$$

```
>>> bin(123)
'0b1111011'
```

Connaissant l'écriture binaire d'un nombre $\langle d_{p-1}, \dots, d_0 \rangle$, on obtient sa valeur en évaluant le polynôme $\sum_{k=0}^{p-1} d_k \mathrm{X}^k$ en 2 :

```
>>> L=[1,1,0,1,1,1,1]
>>> poly(2,L)
123
```

III Preuve d'un programme

1 Problématique

Dans le monde de l'informatique, certains logiciels sont utilisés dans des domaines critiques :
- la santé (production de médicaments, équipements avec radiation, ...) ;
- le transport (pilote automatique d'avion, voiture autonome,...) ;
- les applications militaires ;
- etc.

Pour des raisons de sécurité et fiabilité, il peut être nécessaire de démontrer formellement qu'un programme fait ce pour quoi il a été conçu. Par exemple, une méthode formelle appelée *méthode B* a été utilisée pour valider la programmation de la ligne 14 du métro parisien.

Dans un programme sans boucle avec des instructions conditionnelles, il suffit de procéder à des disjonctions de cas pour dresser l'arbre des possibles et vérifier que l'exécution du code est conforme à sa mission.

Quand un programme utilise une boucle, la preuve formelle devient plus complexe à établir. Deux types de problèmes doivent être résolus :
- la boucle se termine-t-elle ?
- le programme fait-il ce qu'on attend de lui ?

Définition 1. *Un algorithme qui, sous réserve de terminaison, produit le résultat attendu est dit* partiellement correct. *La* correction totale *d'un algorithme est la conjonction de sa terminaison et de sa correction partielle.*

Vocabulaire : Un algorithme est *correct* si la terminaison et la correction partielle de l'algorithme sont satisfaites.

Remarque : On verra ultérieurement un exemple d'algorithme dont ne sait prouver que la correction partielle.

2 Variant de boucle

Pour répondre au problème de l'arrêt d'un programme avec boucle, on utilise un *variant* de boucle.

Définition 2. *Un* variant *de boucle est une variable à valeur dans* $\mathbb{N}$ *et qui décroît strictement à chaque passage dans la boucle.*

Pour un algorithme avec boucle, l'exhibition d'un variant prouve sa terminaison.

• **Recherche du minimum**

```
def mini(L):
    """mini(L:list)->int or float
    Renvoie le minimum de L"""
    n=len(L)
    res=float('inf')
    for k in range(n):
        if res>L[k]:
            res=L[k]
    return res
```

On note n la taille de la liste. La quantité $n-k$ est un variant de boucle : elle décroît strictement et est à valeurs dans $\mathbb{N}$ puisque $k \in [\![0\,;\, n-1]\!]$. Ceci prouve la terminaison du programme.

Dans cet exemple, la terminaison de boucle est triviale car il s'agit d'une boucle inconditionnelle.

• **Recherche d'un élément dans une liste**

```
def detect(elt,L):
    """detect(elt:any,L:list)->bool
    Renvoie test de présence de elt dans L"""
    k,n=0,len(L)
    while k<n and L[k]!=elt:
        k+=1
    return k<n
```

La quantité $n-k$ est un variant de boucle : elle décroît strictement et est à valeurs dans $\mathbb{N}$ du fait de la condition $k < n$ déclenchant la sortie de la boucle. On en déduit la terminaison du programme.

Exercice : La fonction `fact(n)` d'argument `n` entier renvoie la valeur de $n!$.

```
def fact(n):
    """fact(n:int)->int
    Renvoie n!"""
    res=1
    for k in range(2,n+1):
        res*=k
    return res
```

Déterminer un variant de boucle.

Corrigé : La quantité $n-k$ est clairement un variant de boucle.

Exercice : Écrire un programme avec boucle qui, pour un réel p donné, détermine le plus petit entier n tel que

$$\sum_{k=1}^{n} \frac{1}{\sqrt{k}} \geqslant p$$

Préciser un variant de boucle.

Corrigé : On a $\displaystyle\sum_{k=1}^{n} \frac{1}{\sqrt{k}} \geqslant \sum_{k=1}^{n} \frac{1}{\sqrt{n}} = \sqrt{n} \xrightarrow[n\to\infty]{} +\infty$

Ainsi, on peut définir $\forall p \in \mathbb{R} \qquad \varphi(p) = \min\left\{n \in \mathbb{N} \mid \displaystyle\sum_{k=1}^{n} \frac{1}{\sqrt{k}} \geqslant p\right\}$

comme minimum d'une partie non vide de $\mathbb{N}$.

```
def seuil(p):
    k,s=0,0
    while s<p:
        k+=1
        s+=1/np.sqrt(k)
    return s,k
```

La variable `k` compte le nombre de passages dans la boucle `while` et au k-ième passage, la variable s vaut $\displaystyle\sum_{j=1}^{k} \frac{1}{\sqrt{j}}$. Tant qu'on rentre dans la boucle, on a $s < p$ donc $k < \varphi(p)$. La sortie de la boucle est provoquée dès que $s \geqslant p$ d'où $k = \varphi(p)$. Ainsi, la quantité $\varphi(p) - k$ est un variant de boucle.

```
>>> seuil(10)
(33, 10.115589829834331)
>>> seuil(100)
(2574, 100.01870753045409)
```

3 Invariant de boucle

Pour déterminer si un programme avec boucle fait ce pour quoi il est conçu, on utilise un *invariant* de boucle.

Définition 3. *Un* invariant *de boucle est un* prédicat *qui ne change pas au cours de la boucle.*

Vocabulaire : Un *prédicat* est une propriété logique.

Commentaire : En réalité, la notion d'invariant est très vague : on peut imaginer une infinité d'invariants qui n'ont aucune pertinence vis-à-vis d'un programme donné. Dans les faits, quand on parle d'invariant, cela sous-entend un prédicat invariant au cours de la boucle et qui permet de démontrer que le programme fait ce pour quoi il est conçu et prouve donc sa correction partielle.

En pratique, pour établir qu'un prédicat est bien un invariant de boucle, on effectue une démonstration qui est exactement une démonstration par récurrence au sens mathématique usuel : on vérifie que le prédicat est vrai avant l'entrée dans la boucle puis on vérifie qu'il est préservé lors du passage dans la boucle.

• **Calcul du coefficient binomial**

```
def binom(n,k):
    """binom(n:int,k:int)->int
    Renvoie le coefficient binomial k parmi n"""
    res=1
    for i in range(k):
        res=res*(n-i)//(i+1)
    return res
```

Comme invariant de boucle, on peut choisir

$$\mathscr{P}(i) : \quad \texttt{res} = \binom{n}{i}$$

$\mathscr{P}(0)$: Par convention, la propriété $\mathscr{P}(0)$ correspond à l'état des variables avant l'entrée dans la boucle. D'après l'initialisation de `res`, la propriété $\mathscr{P}(0)$ est vraie.

$\mathscr{P}(i) \Longrightarrow \mathscr{P}(i+1)$: Le passage dans la boucle consiste en la transition de `i` à `i+1`. Supposons la propriété vraie au rang i avec $i \in [\![0\,;\, k-2]\!]$ et montrons qu'elle l'est aussi au rang $i+1$. Par hypothèse, on a après passage dans la boucle

$$\texttt{res} = \binom{n}{i} \times \frac{n-i}{i+1} = \frac{n!}{i!(n-i)!} \times \frac{n-i}{i+1} = \frac{n!}{(i+1)!(n-i-1)!} = \binom{n}{i+1}$$

ce qui prouve la propriété au rang $i+1$.

En sortie de boucle, quand $i = k-1$, on a donc $\texttt{res} = \binom{n}{i+1} = \binom{n}{k}$.

• **Recherche du minimum**

```
def mini(L):
    """mini(L:list)->int or float
    Renvoie le minimum de L"""
    res=float('inf')
    for x in L:
        if res>x:
            res=x
    return res
```

On note $\mathrm{L} = [x_0, \ldots, x_{n-1}]$. Comme invariant de boucle, on peut choisir

$$\mathscr{P}(k) : \quad \forall i \in [\![0\,;\, k-1]\!] \qquad \texttt{res} \leqslant x_i$$

avec $k-1$ l'indice de l'élément courant `x` qui parcourt `L`.

$\mathscr{P}(0)$: Par convention, la propriété $\mathscr{P}(0)$ correspond à l'état des variables avant l'entrée dans la boucle. La propriété $\mathscr{P}(0)$ est vraie puisque $[\![0\,;\, -1]\!] = \varnothing$.

$\mathscr{P}(k) \Longrightarrow \mathscr{P}(k+1)$: Le passage dans la boucle consiste en la transition de x_{k-1} à x_k. Supposons la propriété vraie au rang k avec $k \in [\![0\,;\, n-1]\!]$ et montrons qu'elle l'est aussi au rang $k+1$. Si $\texttt{res} \leqslant x_k$, on a

$$\forall i \in [\![0\,;\, k-1]\!] \qquad \texttt{res} \leqslant x_i \quad \text{et} \quad \texttt{res} \leqslant x_k \quad \Longrightarrow \quad \forall i \in [\![0\,;\, k]\!] \qquad \texttt{res} \leqslant x_i$$

Si $\texttt{res} > x_k$, la variable `res` reçoit la valeur de x_k d'où

$$\forall i \in [\![0\,;\, k-1]\!] \qquad \texttt{res} \leqslant x_k \leqslant x_i \quad \Longrightarrow \quad \forall i \in [\![0\,;\, k]\!] \qquad \texttt{res} \leqslant x_i$$

En sortie de boucle, pour $k = n$, le prédicat $\mathscr{P}(n)$ est vrai d'où

$$\forall i \in [\![0\,;\, n-1]\!] \qquad \texttt{res} \leqslant x_i$$

Comme `res` contient une valeur de la liste, il s'agit effectivement du minimum de cette liste.

Exercice : La fonction `fact` réalise le calcul de la factorielle.

```
def fact(n):
    """fact(n:int)->int
    Renvoie n!"""
    res=1
    for k in range(2,n+1):
        res*=k
    return res
```

Déterminer un invariant de boucle.

Corrigé : Comme invariant de boucle, on peut choisir

$$\mathscr{P}(k) : \quad \texttt{res} = k!$$

$\mathscr{P}(0), \mathscr{P}(1)$: On rentre dans la boucle à partir de $k = 2$ donc par convention, les propriétés $\mathscr{P}(0)$ et $\mathscr{P}(1)$ correspondent à l'état des variables avant l'entrée dans la boucle. D'après l'initialisation de la variables `res`, ces propriétés sont vraies.

$\mathscr{P}(k-1) \Longrightarrow \mathscr{P}(k)$: Supposons que la propriété est vraie au rang $k-1$ avec $k \in [\![1\,;\, n]\!]$ et montrons qu'elle l'est aussi au rang k. Par hypothèse, on a après passage dans la boucle

$$\texttt{res} \leftarrow \texttt{res} \times k = (k-1)! \times k = k!$$

En sortie de boucle, pour $k = n$, on a $\texttt{res} = n!$ ce qui est le résultat attendu.

IV Programmation gloutonne

1 Un exemple de rendu de monnaie

On considère un premier système de monnaie formé de jetons de valeurs 5, 2 et 1. Pour rendre la monnaie d'une montant égal à 6, une stratégie dite *gloutonne* consiste à maximiser le rendu à chaque étape. Ainsi, on rend 5 puis 1 et avec deux jetons, on réalise un rendu optimal en le nombre de jetons.

Si l'on considère indifféremment toutes les façons de rendre la monnaie, on rencontre beaucoup de sous-problèmes qui se chevauchent : on rend 1, il reste 5, on rend 1, il reste 4, etc. ou alors on rend 2, il reste 4, etc.

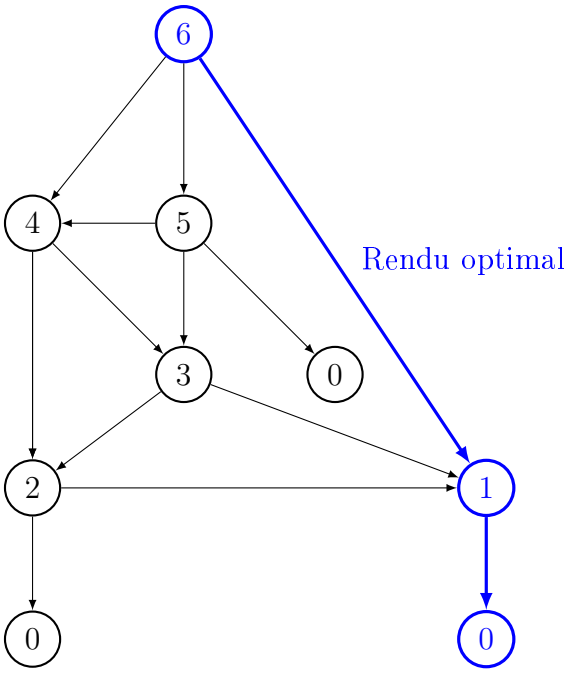

FIGURE 6 – Rendus possibles

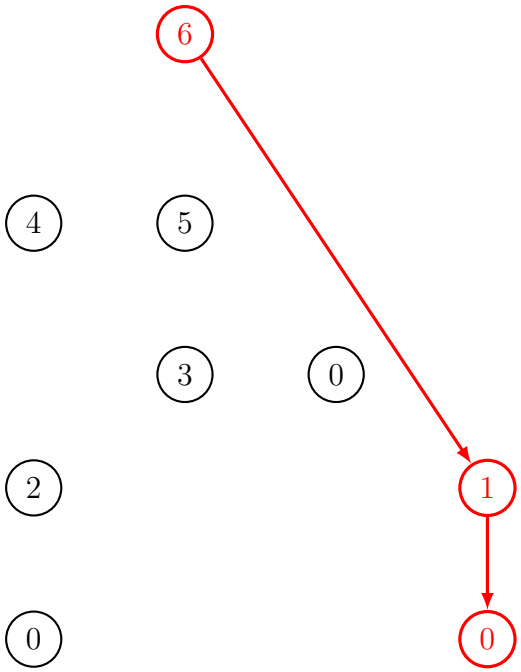

FIGURE 7 – Rendu glouton

On considère ensuite un second système de monnaie formé de jetons de valeurs 4, 3 et 1. Pour rendre la monnaie d'un montant égal à 6, suivant la stratégie gloutonne, on choisit de rendre 4 puis 1 et encore 1. Le premier rendu maximal contraint les rendus suivants et il est aisé de voir que cette stratégie, bien que naturelle, n'est pas optimale : en rendant 3 puis 3 de nouveau, on réalise le rendu avec seulement deux jetons.

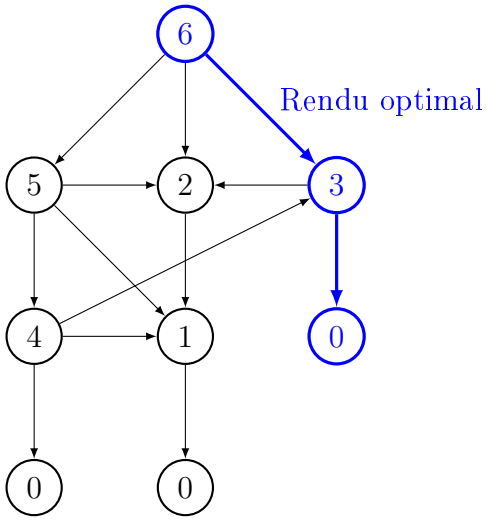

FIGURE 8 – Rendus possibles

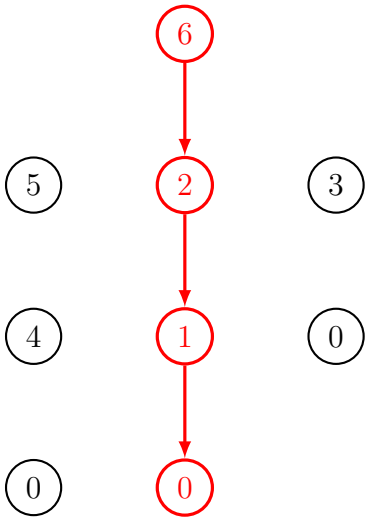

FIGURE 9 – Rendu glouton

2 Principe

Définition 4. *Un algorithme* glouton *(*greedy *en anglais) est un algorithme qui forme une solution en prenant, à chaque étape, le meilleur choix sans remettre en questions les choix précédents (absence de* backtracking*).*

L'approche gloutonne est une démarche simple et intuitive pour proposer une solution à un problème d'optimisation. Mais comme on l'a vu dans l'exemple précédent, rien ne garantit

que la solution gloutonne soit une solution optimale. La *programmation dynamique* permet de déterminer la valeur optimale d'une solution puis de construire une telle solution par une démarche ascendante, avec des sous-problèmes dont la taille croît. Un algorithme glouton choisit une solution particulière par une démarche descendante, avec des sous-problèmes dont la taille décroît.

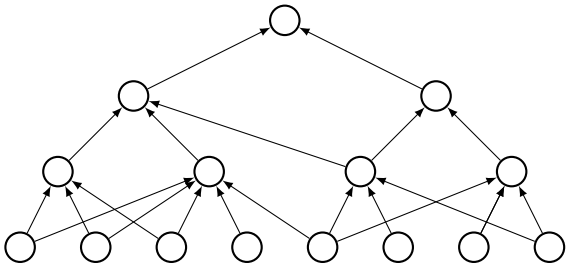

FIGURE 10 – Programmation dynamique

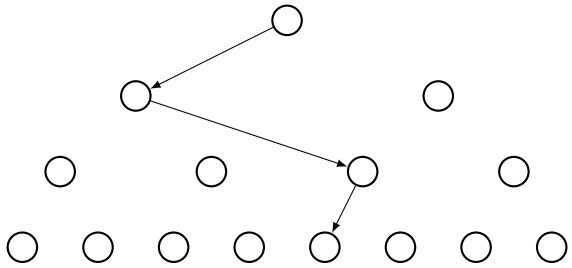

FIGURE 11 – Programmation gloutonne

Face à un problème d'optimisation, il n'y a aucune raison d'avoir confiance en un algorithme glouton ! Même s'il est tentant de proposer une telle approche, celle-ci n'a aucune valeur s'il n'est pas démontré qu'elle fournit une solution optimale au problème. Il y a toutefois des situations célèbres où l'approche gloutonne fonctionne :

- Codage de Huffman ;
- Algorithme de Kruskal ;
- Algorithme de Dijkstra ;
- etc. ...

Le lecteur curieux pourra compléter cette étude en consultant les ouvrages référencés dans la bibliographie.

Références

[1] Jeff Erickson, *Algorithms* , `https://jeffe.cs.illinois.edu/teaching/algorithms/`

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, Third Edition MIT Press and McGraw-Hill, 2009